TRANSPORT LAYER SECURITY (TLS)

Lokesh Phani Bodavula

October 2015

Abstract

1 Introduction

The security of Electronic commerce is completely in the hands of Cryptography. Most of the transactions through e-commerce sites, auction sites, on-line banking, stock trading and many more are exchanged over the network. SSL or TLS are the additional layers that are required in order to obtain authentication, privacy and integrity for all kinds of communication going through network. This paper focuses on the additional layer (TLS) which is responsible for the whole communication. Transport Layer Security is a protocol that is responsible for offering privacy between the communicating applications and their users on Internet. TLS is inserted between the application layer and the network layer-where the session layer is in the OSI model TLS, however, requires a reliable transport channel-typically TCP.

2 History

Instead of the end-to-end argument and the S-HTTP proposal the developers at Netscape Communications introduced an interesting secured connection concept of low-layer and high-layer security. For achieving this type of security there employed a new intermediate layer between the transport layer and the application layer which is called as Secure Sockets Layer (SSL). SSL is the starting stage for the evolution of different transport layer security protocols. Technically SSL protocol is assigned to the transport layer because of its functionality is deeply inter-winded with the one of a transport layer protocol like TCP. Coming to history of Transport layer protocols as soon as the National Center for Supercomputing Application (NCSA) released the first popular Web browser called Mosaic 1.0 in 1993, Netscape Communications started working on SSL protocol. Netscape Communications designed and tried SSL 1.0 with in its organization for finding and fixing bugs. It failed in providing protection for the whole data as it used stream cipher RC4 for data encryption. SSL 1.0 didn’t use sequence numbers. These reasons made SSL 1.0 easily breakable. Altering these flaws
SSL got added with sequence numbers and checksums, but still used an overly simple Cyclic Redundancy Check (CRC).

SSL 1.0 is not released into public officially but Netscape Communications came up with SSL 2.0 in 1994. MD5 is used in SSL 2.0 instead of CRC which is used in SSL 1.0. SSL 2.0 does not follow chain certificate and it uses only RSA key exchange but not different key exchange algorithms. SSL 2.0 has a weak Message Authentication Code (MAC) that uses only 40-bit of encryption in export mode. It is vulnerable in length extension attacks because it uses MD5 hash function. Man-in-Middle attack is easily done in the cipher suit and it is open for truncation attack. It uses a fixed domain certificate with a single service, which will not match with the latest features of virtual hosting for web servers.

Later Microsoft brought out Internet Explorer in 1995 with which it also published a protocol called Private Communication Technology (PCT). PCT is conceptually and technically similar to SSL 2.0. In the same year SSL 3.0 was introduced. SSL 3.0 mainly focused on the shortcomings and security problems in SSL 2.0. SSL 3.0 allows clients and servers to have arbitrary-length certificate chains. SSL 3.0 uses HMAC, which is more powerful than MAC and it uses 128-bit of encryption. The record or information cannot be altered by attacker even sending on open private network. Different keys are used in SSL 3.0, and hence even if weak ciphers are used, mounting attacks against message authenticity and integrity can still be made intractable by using long keys for message authentication. SSL 3.0 can defend against Man-in-Middle attack by keeping the authenticated finished message with including a hash for all the previous handshake messages. SSL 3.0 has general key exchange protocol and it permits the Diffie-Hellman and Fortezza key exchanges and non-RSA certificates. It provides chain certificates for client and server. It uses SHA-1 hashing algorithm, which is more secure than MD5 algorithm. It supports extra cipher suits.

Both Netscape Communications and Microsoft created confusion among the security community by encouraging their products SSL 3.0 and PCT. Microsoft took forward step and proposed a new protocol called Secure Transport Layer Protocol (STLP) that was basically a modification of SSL 3.0. In 1996 an IETF Transport Layer Security (TLS) Working Group was formed to resolve the issue and standardize a unified TLS protocol. Finally after three years of struggle IETF TLS WG released TLS 1.0 which was specified in RFC 2246. There are very less differences between SSL 3.0 and TLS 1.0. The key generation functions and MACs in TLS 1.0 and SSL 3.0 are different. TLS requires DSS/DH support and it has more alerts. Later in 2006 RFC 2246 is made obsolete by specifying TLS 1.1 in standards Track RFC 4346.
In 2008, the TLS protocol version 1.2 was specified in standards track RFC 5246. In TLS 1.2 the MD5/SHA-1 combination in the Pseudo-Random Function (PRF) was substituted with cipher-suite-specified PRFs. The MD5/SHA-1 combination in the digitally-signed element was replaced with a single hash. Signed elements include a field explicitly specifying the hash algorithm used. There is added support for authenticated encryption with added data modes. To determine the type of hash and signature algorithms there was substantial cleanup to the client’s and server’s ability. Apart from these developments there are further more developments which made TLS 1.2 as the advanced version of TLS 1.1. Presently TLS 1.2 is in use and as of September 2015, TLS 1.3 is a working draft.

3 TLS Handshake Protocol:

The TLS Handshake protocol is one of the high-level clients of the TLS Record Protocol. The negotiations are done by this protocol using secure attributes of a session. These handshake messages are sent to the TLS record layer, here they are encapsulated into one or more structures such as the TLSPlaintext and these are processed and transmitted by the currently active session state.

Initially client and server exchanges messages to establish security enhancement capabilities between them. Whenever the ClientHello message is send by client to server, the Server should reply with ServerHello message. Else fatal error occur which results in failure of connection. The ClientHello and ServerHello establish the following attributes: Protocol Version, Session ID, Cipher Suite and Compression Method. Additionally two random values are generated and exchanged: ClientHello.random and ServerHello.random. The four messages that key exchange use are: the client certificate, the ClientKeyExchange, the server Certificate and the ServerKeyExchange.

After hello messages, the server sends a Certificate message for authentication which consists of its certificate. In case if server is already authenticated it may request certificate from client. Server sends ServerHelloDone message to end the hello message phase of handshake. The server will then wait for client response and if the server has sent a CertificateRequest message, the client must send the Certificate message. Now the ClientKeyExchange message is sent. The content of this message will depend on the public key algorithm selected between the ClientHello and ServerHello. Client copies the pending Cipher Spec into the current Cipher Spec when it sends the ChangeCipherSpec message. Immediately the client sends finished message under the new algorithms, keys and secrets. After this processing a response is sent by the server, the message ChangeCipher Spec is transferred. This pending message is transferred to the current Cipher Spec and the finished message is sent under the new Current Spec. Thus, the handshake technique is now complete and the data can be exchanged through the application layer between the client and the server and this data must not be sent prior to the first handshake completion.
4 Hello Messages:

4.1 Hello Request
HelloRequest is used to initiate a new negotiation process. Client starts the negotiation process upon receiving the HelloRequest and sends a ClientHello message. If client is busy it responds with no renegotiation alert. Server results fatal alert whenever it doesn’t get any reply for its HelloRequest from client with ClientHello. The structure of the HelloRequest is ‘struct { } HelloRequest’.

4.2 Client Hello
ClientHello is the message that a client has to send server saying that it is connected and is ready for negotiation process. ClientHello message can be on its own or a response to server’s HelloRequest. The structure of ClientHello is

```
struct {
    uint32 gmt_unix_time;
    opaque random_bytes[28];
} Random;
```

4.3 Server Hello
The server sends this message as a response to the ClientHello message when it is able to find acceptable set of algorithms. If the match is not found, the response of the handshake will be a failure alert. The Structure for this message is as follows:

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    }
} ServerHello;
```

The extensions can be detected by the bytes that determined in the following compression_method at the end of the ServerHello.

- Server version: This field contains the lower of the suggested by the client in the client hellow and the highest that is supported by the server.
• Random: The structure that is generated by the server and which must be independently generated by the ClientHello.random.

• Session_id: For the identity of the session corresponding to the connection, the ClientHello.session_id is empty and hence the session cache is checked for the match. When the match is found the server establishes a new connection using a specified session state, server then responds with the same value as the clients’ applied. If this field contains a different value identifying the new session, the server returns the value as an empty session_id and the session is not cached and is not resumed.

• Cipher_suite: The cipher selected by the server from the list in ClientHello.cipherSuites. This field is resumed value from the state of session.

• Compression_method: The compression algorithm is selected from the list of ClientHello.compression_methods. This is the value from the field of resumed session state.

• Extension: The extensions are offered by the client are only present in the server’s list.

5 Server Certificate

Server sends the Certificate message followed by ServerHello message which consists of server certificates chain upon agreeing the key exchange method. The certificates send by server should be acceptable for the negotiated cipher suite’s key exchange algorithm and any negotiated extensions. The structure of the certificate message is

```plaintext
opaque ASN.1Cert<1..2^24-1>;

struct {
    ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
```

tificate message is

6 Server Key Exchange

The Server Key Exchange Message transfers the cryptographic information to allow the client to communicate premaster secret. Certificate message is immediately followed by Server Key Exchange Message. ServerKeyExchange message comes into existence when the server Certificate message does not contain enough data for allowing client to exchange premaster secret. This is true for DHE_DSS, DHE_RSA and DH_anon key exchange methods. The structure of this message is
enum { dhe_dss, dhe_rsa, dh_anon, rsa, dh_dss, dh_rsa
        /* may be extended, e.g., for ECDH -- see [TLSECC] */
    } KeyExchangeAlgorithm;

struct {
    opaque dh_p<1..2^16-1>;
    opaque dh_g<1..2^16-1>;
    opaque dh_Ys<1..2^16-1>;
} ServerDHPrams;    /* Ephemeral DH parameters */

dh_p
    The prime modulus used for the Diffie-Hellman operation.

dh_g
    The generator used for the Diffie-Hellman operation.

dh_Ys
    The server's Diffie-Hellman public value (g^X mod p).

struct {
    select (KeyExchangeAlgorithm) {
        case dh_anon:
            ServerDHPrams params;
        case dhe_dss:
        case dhe_rsa:
            ServerDHPrams params;
            digitally-signed struct {
                opaque client_random[32];
                opaque server_random[32];
                ServerDHPrams params;
            } signed_params;
        case rsa:
        case dh_dss:
        case dh_rsa:
            struct {} ;
        /* message is omitted for rsa, dh_dss, and dh_rsa */
        /* may be extended, e.g., for ECDH -- see [TLSECC] */
    }
} ServerKeyExchange;
7 Server Hello Done

The ServerHelloDone message indicates that the server has sent all the supporting messages to support the key exchange. The client should verify the certificate of server upon reception of ServerHelloDone message. The structure of ServerHelloDone message is `struct {} ServerHelloDone;`.

8 Client Certificate

Server uses the Client certificate message while verifying the CertificateVerify message to check the client’s certificates. After ServerHelloDone message from server client can send the client certificate if server requests a certificate. It is upto server whether to continue with client without certificate request or respond with a fatal handshake_failure alert if client got no certificates.

9 Client Key Exchange

The premaster secret is set with this message either by transmission of RSA-encrypted secret or by the transmission of Diffie-Hellman parameters that will allow either side to negotiate upon same premaster secret. If the certificate from client is containing a static DH exponent then this message should be sent and should be null. If ephemeral Diffie-Hellman exponent is used by client then this message contains the client’s Diffie-Hellman public value. The structure of this message is

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa:
            EncryptedPreMasterSecret;
        case dhe_dss:
        case dhe_rsa:
        case dh_dss:
        case dh_rsa:
        case dh_anon:
            ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

10 Certificate Verify

This message is used to provide the exact verification of client certificate. The structure of this message is
11 Finished Message

Once the finished message has sent, received and validated by an end then everything is clear to share the application data. The finished message is tested by the algorithms, keys and secrets. The structure of this message is

```c
struct {
  digitally-signed struct {
    opaque handshake_messages[handshake_messages_length];
  }
} CertificateVerify;
```

The finished label for the finished messages sent by the client the string used is “client finished” and for the finished messages sent by the server the string used is “server finished”. Hash of the handshake message is denoted by hash. A fatal error occurs if a finished message is not preceded by a ChangeCipherSpec message at appropriate point in the handshake.

12 Front end and Back end (My observation and conclusion)

Here the back end process refers to the communication to negotiate the security parameters between the client and server. The summary of back end process that takes place in handshaking is shown in below figure.

```
Client                        Server
ClientHello                ServerHello
Certificate*           Certificate
ServerKeyExchange*     ServerKeyExchange*
CertificateRequest*     CertificateRequest*
<-----                  <-----
ServerHelloDone  ServerHelloDone

Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished            [ChangeCipherSpec]
<----->               <----->
Application Data     Application Data
```

Figure: Message flow for a full handshake

Analogous to the back end process is translated in the front end (on browser)
to notify the user what actually is going in the handshake process. The details about a website, our connection to it and our visit history are notified for the user so that a user can know whether he is established with a secure connection with the site or not. Whenever we go to a URL an icon appears in the top left of the address bar. These icons give us information about site identity and connection.

12.1 Site identity information

The icon that appears gives information about a site whether it is using Transport Layer Security (TLS) or Secure Sockets Layer (SSL) certificates. A site is genuine if it has a valid TLS/SSL certificate and if some invalid certificates appear it means that someone is trying to tamper our connection to site which are also known as Man-in-Middle attacks. If a green lock icon appears in the address bar then it means that the site’s certificate is valid and its identity has been verified by a trusted third party. If an orange exclamation icon appears that means no certificate is provided by the site. If a lock icon with yellow warning triangle occurs that means proceed with caution. There are common mistakes in website’s configurations, but that does not guarantee that the connection is secure. If red lock icon appears it means that there are problems with site’s certificate. If we disclose any information on the website, the person on the network might be able to read it.

12.2 Connection information

These icons let us know whether our browser has a secure connection with the site you’re on. The green lock icon represents a secure connection. The orange exclamation icon represents that the connection to site is not encrypted. The lock icon with yellow warning triangle represents the unwanted images or ads on our page. The red lock icon warns us to be careful if we’re entering personal information on that particular page.

13 Conclusion

The handshaking process is a virtual process between the client and server. But a user can view the results of the handshaking process on his browser. Whenever a connection started a hello messages phase starts in the back end. In this hello message phase there will be verification of certificate and selection of key exchange techniques. The results of the back end process are represented in address bar.
In the above figure the result of handshake certifies that the site requested is trusted which means all the certificates are verified. The connection used is TLS 1.2 and it is encrypted with 256-bit encryption. Whenever a handshake process results with a red lock it means that there is problem in verifying the certificates.